Australian Government
Department of Defence
Defence Science and
Technology Organisation

The Role of Formal Methods in High-Grade InfoSec Evaluations

Benjamin W. Long
School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, Australia 4072

Information Networks Division
Defence Science and Technology Organisation

DSTO–GD–0456

ABSTRACT

With the increasing use of computer systems in governmental, commercial and industrial equipment, we must be sure that these systems remain secure. The Common Criteria is an internationally recognised criteria for evaluating IT products with security functionality. To achieve a high level of assurance from the Common Criteria, formal methods should be applied in the development process. This report concentrates on formal methods support for development and evaluation of security-critical systems in the Common Criteria. In particular, the Defence Signal Directorate (DSD) is charged with the oversight of the security evaluations program in Australia. The report attempts to indicate what DSD should know about formal methods for the high-grade evaluations, and where they can find out more as desired.

Published by

Defence Science and Technology Organisation
PO Box 1500
Edinburgh, South Australia 5111, Australia

Telephone:	*(08) 8259 5555*
Facsimile:	*(08) 8259 6567*

AR No. 013-595
March, 2007

The Role of Formal Methods in High-Grade InfoSec Evaluations

EXECUTIVE SUMMARY

This report was prepared for AS InfoSec DSD by the University of Queensland under contract from the DSTO (arising from task JW 02/106). The aim of the contract was to conduct a survey of formal verification methodologies and how those methodologies relate to the Common Criteria for high assurance evaluation. The report is aimed at those readers who are familiar with development/certification procedures, but are not especially familiar with formal methods and their role in that process. In particular, the Defence Signal Directorate (DSD) is charged with the oversight of the security evaluations program in Australia. The report attempts to indicate what DSD should know about formal methods for the high-grade evaluations, and where they can find out more as desired.

The *Common Criteria* is an internationally recognised set of criteria for evaluating IT products with security functionality. The criteria defines seven *Evaluation Assurance Levels* which identify increasing levels of assurance and the requirements that must be satisfied by a product in order to achieve them. For a product to achieve a high level of assurance the criteria demands strict application of *formal methods* to the development process.

Part 2 provides a brief review of the developmental requirements outlined by the Common Criteria. This highlights where the requirements for formal analysis enter, but also should help to clarify the actual statements of requirement. A selection of comments on the Common Criteria are given in Part 3, together with a brief discussion of the tool support which would be ideal in meeting the formal requirements for high-grade evaluations.

In Part 4 we discuss formal specification methods. We take the view that there are two main candidates for non-trivial systems, which differ in whether one primarily focusses on the events the system can engage in or the states through which it can evolve. We try to expose their relative merits through examples. The combination of the two is possibly interesting as a research area.

Having specified the system and desired critical properties, the next step is to verify that the system model indeed satisfies those properties. Again there are a number of alternative methods available, depending largely on the degree of automation compared to the degree of user interaction. Part 5 explains the trade-off made when a given method is chosen.

Finally, Part 6 lists the formal methods which have been used by various organisations when developing security products to high levels of assurance. It is interesting to note that there seems to be little supporting documentation freely available in the public domain. At least in part this is due to commercial confidentiality.

Author

Benjamin W. Long
The University of Queensland

Benjamin Long is currently doing a PhD at the University of Queensland in which he is investigating the merits of the Z specification language and SAL model checker for a formal approach to security protocol analysis.

Contents

Part 1

Introduction

With increasing use of computer systems in governmental, commercial, and industrial equipment, we must be sure that these systems remain secure. A secure computer system provides three fundamental properties [46]:

- *confidentiality* — information is disclosed only according to policy;
- *integrity* — information is not destroyed or corrupted and the system performs correctly; and
- *availability* — system services are available when they are needed.

For example, a governmental system vulnerable to a breach of confidentiality could be hacked by an intruder, allowing the intruder to learn a vital stratagem; a commercial bank system vulnerable to a breach of integrity could process bogus transactions that benefit or disfavour account holders; and combat systems attacked by a destructive intruder could suffer from a lack of availability and might not be able to give a timely command in order to release a defensive projectile.

There are numerous products available that aim to achieve such security properties through various *security enforcing functions.* For example, a *data diode* ensures that information travels in only one direction on a communication channel, thus preventing unwanted 'leakage' of information in the opposite direction. Users of such products need to know that they successfully implement the desired security function without introducing any new vulnerabilities. To cater for this need, assurance is attainable from *certification bodies* whose purpose is to evaluate security products against given *security evaluation criteria.*

In the late 1990s, the United States, together with the European Union and Canada, released the *Common Criteria* (CC) [1], an internationally recognised set of criteria for evaluating IT products with security functionality. It was based on criteria (TCSEC [2], ITSEC [3], and CTCPEC [4]) previously devised by the co-operating parties. The CC defines seven *Evaluation Assurance Levels* (EAL1 through EAL7), which identify increasing levels of assurance and the requirements that must be satisfied by a product, or *target Of Evaluation* (TOE), in order to achieve them. For TOEs to achieve high EALs, the CC demands strict application of *formal methods* to the development process.

The term "Formal methods" covers quite a range of possible techniques. Underlying them all is the use of mathematically well-defined languages to capture the system design, and a consistent logic in which to derive its consequences abstractly. Through this medium the developer is forced to confront the problem of describing the system unambiguously in a language which manifestly exposes the failure to do so. Moreover, abstract reasoning can be considered as pattern-matching each step in the derivation to a consistent rule set, and therefore – at least when enforced through proof tools – does not allow unstated contextual information or prejudices to influence the analysis. Thus, there are two immediate benefits of the use of formal methods in any development process:

1 increased understanding of the system, at an early enough stage so that inadequacies can be corrected cost-effectively; and

2 assurance that any critical properties which have been stated and proved can be trusted.

Employing formal methods tools in the certification process brings up another benefit:

3 the evaluator has a complete treatment of the required analysis which can be checked to be consistent by the tool, and thus can concentrate on the completeness of the modelling and verification regime.

Clearly there is much to be gained from the simple requirement of mathematical precision and consistency.

It is not at all surprising, then, that the use of formal methods is mandated in preparing high-grade evaluations. In particular, there are two settings where the CC requires the application of formal methods:

1 formal specifications are required for producing precise unambiguous descriptions of the TOE's behavioural properties; and

2 formal verification is required to prove a correspondence between two or more of the specifications, providing convincing evidence that critical functionality is implemented through the entire development process.

The actual requirement depends on the EAL sought.

This report concentrates on formal methods support for development and evaluation of security-critical systems in the Common Criteria. It is aimed at those readers who know about development/certification procedures, but are not especially familiar with formal methods or their role in that process. In particular, the Defence Signal Directorate (DSD) is charged with the oversight of the security evaluations program in Australia. The report attempts to indicate what DSD should know about formal methods for the high-grade evaluations, and where they can find out more as desired.

Part 2 provides a brief review of the developmental requirements outlined by the Common Criteria. This highlights where the requirements for formal analysis enter, but also should help to clarify the actual statements of requirement. A selection of comments on the Common Criteria are give in Part 3, together with a brief discussion of the tool support which would be ideal in meeting the formal requirements for high-grade evaluations.

In Part 4 we discuss formal specification methods. We take the view that there are two main candidates for non-trivial systems, which differ in whether one primarily focusses on the events the system can engage in or the states through which it can evolve. We try to expose their relative merits through examples. The combination of the two is possibly interesting as a research area.

Having specified the system and desired critical properties, the next step is to verify that the system model indeed satisfies those properties. Again there are a number of alternative methods available, depending largely on the degree of automation compared to the degree of user interaction. Part 5 explains the trade-off made when a given method is chosen.

Finally, Part 6 lists the formal methods which have been used by various organisations when developing security products to high levels of assurance. It is interesting to note that there seems to be little supporting documentation freely available in the public domain. At least in part this is due to commercial confidentiality.

Part 2

Common Criteria: Development Requirements

A CC evaluation of a TOE depends on its conformance to a specific set of security requirements. The requirements to which the TOE is to conform are firstly devised and presented within a document called a *Security Target* (ST) which is evaluated to demonstrate that it is complete, consistent and technically sound. It is then suitable as the basis for the corresponding TOE evaluation.

Security requirements are divided into several *classes*. The class containing *assurance requirements* associated with the development process of the TOE is the ADV class [1] (see also Part 2.2. Within this class are requirements demanding certain levels of formality to be used for various stages of development.

Section 2.1 introduces fundamental CC terminology used within the security target, Section 2.2 summarises the requirements contained in the ADV class, Section 2.3 defines what is meant by the different levels of formality discussed within the ADV class, and Section 2.4 discusses the current state of play for following the criteria set out in the CC and presents areas still open for research.

2.1 Security Target

As seen in Figure 1 (which appears in Part 2 of the CC), a security target for a TOE contains definitions of the *security environment*, *security objectives* and *security requirements*, and a *summary specification* [1] as follows.

- The security environment defines the context in which the TOE is intended to be used and identifies assumptions, threats, and organisational security policies.
- The security objectives address all of the concerns raised from analysis of the security environment and determine whether they are addressed directly by the TOE or by its environment.
- Security requirements are refined from the security objectives which, if met, will ensure that the TOE can meet its objectives.

Security requirements are stated as both *functional requirements* and *assurance requirements*:

– functional requirements identify what the *system* must do about ensuring security in terms of a system's technical features and capabilities; whereas

– assurance requirements identify what *developers* must do to prove that the system really does what is intended in terms of procedures to be performed and documents to be prepared [46, 42].

The *TOE Security Policy* (TSP) is essentially a collective name for a collection of these requirements. Note that in this part the ST can optionally state security requirements for the IT environment.

- The summary specification provides a high-level definition of the *TOE Security Functions* (TSF) claimed to meet the functional requirements, and assurance measures taken to meet the assurance requirements.

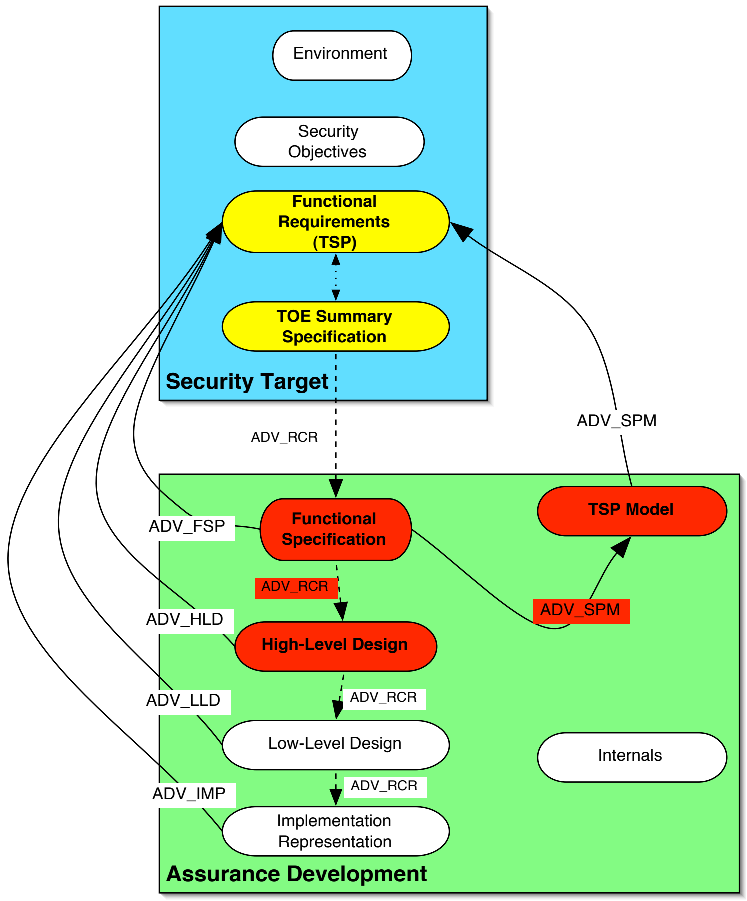

Figure 1: *The ST and ADV and flows between*

2.2 The ADV class

The ADV assurance class [1] associated with the development process of the TOE defines requirements for the stepwise refinement of the TSF from the TOE summary specification in the ST down to actual implementation. The ADV class is divided into seven families of requirements, again seen in Figure 1, each one corresponding to one of the following parts of the development process.

- The *Functional Specification* describes the TSF, and must be a complete and accurate instantiation of the TOE security functional requirements.
- The *High-Level Design* is a top level design specification that refines the TSF functional specification into the major constituent parts of the TSF. It identifies the basic structure of the TSF and the major hardware, firmware, and software elements.
- The *Low-Level Design* is a detailed design specification that refines the high-level design into a level of detail that can be used as a basis for programming and/or hardware construction.
- The *Implementation Representation* is the least abstract representation of the TSF. It captures the detailed internal workings of the TSF in terms of source code, hardware drawings, etc., as applicable.
- The *Representation Correspondence* is a demonstration of mappings between all adjacent pairs of available TSF representations, from the TOE summary specification through to the least abstract TSF representation that is provided.
- The *TSF Internals* requirements specify the requisite internal structuring of the TSF.
- The *TOE Security Policy Model* is a structured representation of security policies of the TSP, and is used to provide increased assurance that the functional specification enforces the security policies of the TSP – and, ultimately, the TOE security functional requirements. This is achieved by correspondence mappings between the functional specification, the security policy model, and the security policies that are modelled.

Table 1 lists all components from the seven families of requirements, and identifies which component (if any) from each family is needed to fulfil the assurance requirements for achieving a particular EAL.

2.3 Levels of Formality

The functional specification, high-level design, low-level design, and the TSP models are written using one or more of three specification styles: *informal*, *semiformal* or *formal*. These levels of formality are defined in the CC as follows.

		EAL1	EAL2	EAL3	EAL4	EAL5	EAL6	EAL7
FSP:	Informal functional specification	●	●	●				
	Fully defined external interfaces				●			
	Semiformal functional specification					●	●	
	Formal functional specification							●
HLD:	Descriptive high-level design		●					
	Security enforcing high-level design			●	●			
	Semiformal high-level design					●		
	Semiformal high-level explanation						●	
	Formal high-level design							●
LLD:	Descriptive low-level design				●	●		
	Semiformal low-level design						●	●
	Formal low-level design							
IMP:	Subset of the implementation of the TSF				●			
	Implementation of the TSF					●		
	Structured implementation of the TSF						●	●
RCR:	Informal correspondence demonstration	●	●	●	●			
	Semiformal correspondence demonstration					●	●	
	Formal correspondence demonstration							●
Internals:	Modularity					●		
	Reduction of complexity						●	
	Minimisation of complexity							●
SPM:	Informal TOE security policy model				●			
	Semiformal TOE security policy model							
	Formal TOE security policy model					●	●	●

Table 1: *ADV components for achieving particular EALs.*

- An informal specification is written in natural language prose, that is, a commonly spoken tongue used for communication among human beings; for example, English.
- A semiformal specification is written in a restricted syntax language, usually accompanied by supporting explanatory (informal) prose. The restricted syntax language may be a natural language with restricted sentence structure and keywords with special meanings, or it may be diagrammatic: for example, it could make use of data-flow diagrams, state transition diagrams, entity-relationship diagrams, data structure diagrams, and process or program structure diagrams.
- A formal specification is written in a notation based upon well-established mathematical concepts, usually accompanied by supporting explanatory (informal) prose. The notation should have a well-defined syntax and semantics and well-established proof rules that support logical reasoning.

Correspondence mappings are required to prove that each of the TSF representations, from the functional specification through to the implementation, correspond to each other;

and for the security policy modelling to ensure that the TSP model corresponds to the functional specification. A correspondence can take the form of an informal demonstration, a semiformal demonstration, or a formal proof.

- Informal demonstration of correspondence demands only basic evidence, for example, checklist tables or appropriate notation of design diagrams.
- Semiformal demonstration of correspondence requires a more structured approach than is needed for an informal demonstration, lessening ambiguity that could exist in an informal correspondence.
- Formal proof of correspondence is achieved by reasoning with the formal specifications with appropriate application of suitable proof rules. In order to verify correspondence of the TSP model, desirable security properties must be expressible in the chosen formal specification language and these properties need to be shown to be satisfied by the formal specification.

2.4 Discussion

General guidance for evaluation of security products under the CC is provided within the Common Methodology for Information Technology Security Evaluation (CEM). It describes the minimum actions to be performed by an evaluator in order to conduct a CC evaluation. The CEM currently provides guidance for evaluations against levels EAL1 through EAL4 only, with no guidance for high level evaluations requiring formal methods. This is the guidance followed by all countries involved in the Common Criteria Recognition Arrangement (CCRA) which was established to regulate evaluations ensuring that co-operating countries perform evaluations to the same level of rigour.

Australia and New Zealand are both part of the CCRA and follow an additional program, the Australasian Information Security Evaluation Program (AISEP) [5], that provides more specific guidance as to how security evaluations are to be achieved given the organisational structures in place within Australia and New Zealand. It was first introduced for use within Australia in 1994, to guide compliance with the requirements set out in the ITSEC [3]. However, since then New Zealand has joined the program and guidance for the Common Criteria has also been incorporated.

Although evaluation levels EAL5 through EAL7 are not included in the CCRA, all levels of the ITSEC *are* mutually recognised between Australia, New Zealand, and the United Kingdom. Therefore, any security products requiring high assurance evaluations under the AISEP are currently performed against ITSEC levels E4 to E6. The University of Queensland is currently researching ways to improve the DSD's internal evaluation procedures, and hopes to investigate methodologies for CC high assurance evaluations that require the use of formal methods for use within the AISEP in the near future.

Part 3

Formal Analysis for the Common Criteria

The CC's formal requirements for high-assurance evaluations can be read from Table 1. They are all concentrated in the ADV class, and can be summarised as follows (names refer to those used in Figure 1).

- Functional Specification: formal specification of the TSF and corresponding external interfaces: for the interfaces the specification must detail all effects, exceptions and error messages.
- High-Level Design: a formal model which functionally decomposes the TSF into subsystems with well-defined security functionality – an interface specification, with details as before, is required for each sub-component.
- TSP model: a formal model of the security properties – of all "rules and characteristics" of the TSP.
- ADV_RCR: formal verification that the High-Level Design is a refinement of the Functional Specification.
- ADV_SPM: formal verification that the Functional Specification enforces the requirements of the TSP Model.

Thus we find that there are three major areas of formal analysis required by the CC to be covered in such an evaluation: *specification*, *modelling*, and *verification*. In later parts we develop some of the theoretical notions contained[1] under these headings. In this Part we comment on the requirements themselves, and on the nature of "ideal" tool support for the formal analysis.

[1]Actually we will consider the distinction between specification and modelling as a question of detail, and simply discuss them together.

3.1 General comments on the CC

The CC has some looseness in its language, which would have to be tied down in any given evaluation. That is not necessarily a bad thing, probably anticipating a wide range of possible applications of the process. Here we briefly mention a few examples.

In Section 2.1 we have identified the TOE Security Policy as the collection of functional requirements. Indeed, the TSP does not seem to have status as an independent entity in the CC, despite the statement that the primary purpose for TOE evaluation is to ensure that the TSP is enforced over the TOE resources. Rather, the CC elsewhere states that:

> "The developer is not explicitly required to provide a TSP, as the TSP is expressed by the TOE security functional requirements, through a combination of Security Function Policies (SFPs) and the other individual requirement elements."

There are two points to be made here.

1. The SFPs alluded to here are contained in Chapter 2 of the CC, at least a number relating to either access control or information flow control, together with a list of functions which can be used to enforce them.

   The "preferred" approach would be to construct the TSP from the listed SFPs. However, it is anticipated that the list may not be sufficient, and in Chapter 1 Annex A of the CC we find

   > "Should none of the Chapter 2 or Chapter 3 requirements components be readily applicable to all or part of the security requirements, the ST may state those requirements explicitly without reference to the CC."

   The only advice here seems to be that

   > "Any explicit statement of TOE security functional or assurance requirements shall be clearly and unambiguously expressed such that evaluation and demonstration of compliance is feasible. The level of detail and manner of expression of existing CC functional or assurance requirements shall be used as a model."

   It is expected that the actual statements would be developed in consultation with the evaluators.

2. The "other individual requirement elements" referred to here can surely only be IT-environment requirements. It is unlikely that assurance requirements should form part of the TSP, since they are meta-level entailments on the process rather than the system/product being developed. In fact, the assurance requirements can be stated simply as the choice of EAL sought. On the other hand, it is quite reasonable that environmental requirements should be part of the TSP: since the TOE will not be operating in a vacuum there are certain physical assumptions whose exposure

could enhance the understanding of the security provided. Again it is expected that consultation with the evaluators will ensure that the security of the actual TOE is not compromised by the nature of these additional requirements.

The way in which the evaluator is implicitly drawn into the development process is an interesting aspect of the CC. In practice it must clearly be regulated to ensure the ultimate independence of the assessment. In the Australian context, the AISEP is ideally suited to play this oversight role.

In Figure 1 the flows included in ADV_FSP, ADV_HLD, ADV_LLD and ADV_IMP – from the ADV hierarchy back to the ST requirements (or we can say, the TSP) – are necessarily "informal" since the ST is also informal The CC seems to state these as requirements on the evaluator; e.g.,

> "The evaluator shall determine that the functional specification is an accurate and complete instantiation of the TOE security functional requirements."

Presumably such statements imply that the evidence for the determination should be explicitly presented to the evaluators. It is not immediately clear that added assurance is gained from these repeated comparisons to the TSP. Referring to Figure 1, the CC effectively requires a refinement down the vertical column.

- In the (dotted) relation between parts 3 and 4 of the ST the developer must show
  - coverage: all security requirements are enforced by some TSFs, and all TSFs are required to completely enforce those requirements; and,
  - satisfaction: the security requirements are indeed met by the TSF.
- The ADV_FSP must be a complete representation of TSF.
- Flows down the TSF representation hierarchy comprising the ADV are refinements.

In a strict sense then, the proof required in ADV_SPM that the TSP is enforced by the functional specification gives assurance that the lower levels of refinement enforce the TSP. This question leads one to speculate on whether the security functionality can be entirely presented at the highest level.

The word "correspondence" is over-loaded in the CC, but often in the final listing of requirements of the different ADV layers it is replaced by a more precise word. This is not done for the ADV_SPM, but we have replaced it by "enforce" since there does not seem to be any other correspondence worth proving.

The final general comment is important. The CC requirement in the ADV_SPM is to show that the TSP Model completely models the TSP "with respect to all policies of the TSP that can be modelled". In fact, earlier the CC states that

> "modelling certain policies is currently beyond the state of the art."

It is not clear what this might mean, or whether it is EAL-dependent, and presumably again it is the job of the evaluators to ensure that such a statement is not invoked without good reason. Still, it seems worthwhile to stress here the opposing view that "if it can't be modelled it isn't a security property"! Modelling is not an onerous task, but rather a transcription to a mathematically precise language of one's understanding. Being "not amenable to modelling" then translates to being "not sufficiently understood". Certainly for high-grade evaluations we urge that statements which are not understood cannot serve as security policies.

3.2 Tool support for formal analysis

Our aim in this section is to consider the formal requirements of the CC from the point of view of what tool support would be ideal. Further comments on the CC arise from these considerations. Any discussion of actual tools currently available will be left for later sections.

The CC anticipates that the TOE will have a variety of functional components. Even if the actual security functions are simple it could be quite complex to explain the entire system. An ideal tool would certainly have a specification/modelling language which:

- includes propositional logic for reasoning;
- should be flexible and even extensible in principle to allow the description to be apt for the given TOE;
- can reference system components in a hierarchically sensible way;
- contains at least temporal notions;
- for precision may allow explicit time/time interval references; and
- has an expressive syntax which is neatly presented in the tool's documentation.

It would be of particular value if the language was "seamless" from specification to model: i.e., the objects naturally discussed in the specification can be elaborated further. This means that those objects have some dynamical status in the language, and their actual dynamics can be revealed as the description becomes more detailed. Thus, the language in the ideal tool should allow for:

- standard formulations of system dynamics, such as state machines; and
- mechanisms to smoothly evolve the system description from a high-level to a more detailed dynamical level.

There are two major paradigms for specification and modelling used in the formal methods community.

1. In *event-based* approaches the system is modelled as communicating processes which can themselves be understood in terms of the sequence of events (actions) in which they can partake. The communication is through synchronisation on shared events. Security properties can be represented as predicates which determine allowed traces. Alternatively, the property may itself be represented as a process, any implementation of which will be acceptable as an implementation of the system.

2. As the name would suggest, in *state-based* approaches the system state (the "memory" as well as inputs and outputs) is explicitly modelled, and system actions are then modelled as functions on state. Typically such models would have explicit variables which encode the state data. The actual representation of state depends on the nature of the problem: e.g., for incorporating temporal notions one could represent the state as a sequence capturing successive snapshots of the configuration held by the system variables. Security properties can be represented as state predicates.

Not surprisingly, event-based approaches are suited to systems which look like communicating processes – as we will discuss later. In fact, since the essence of the description is in terms of sequences of system actions, these approaches provide a simple specification of the order of critical events. Thus, where the order of events is crucial, this can be effectively applied as a specification language.

State-based approaches, however, are more generic, and have the added advantage of incorporating a natural attack on any mathematical or physical problem: clarity ensues from carefully stating the variables, and the functions between them. The properties when written down can then be understood almost by inspection. In contrast, when a generic system is forced to fit the communicating processes paradigm for the actual system model, then the process description of properties can be quite esoteric. Moreover, even though the analysis of systems which look like communicating processes can be streamlined, the apparently unphysical modelling with interleaving and instantaneous synchronisation is itself a barrier to system understanding. Although the hook-up of sub-components in the state-based approach can be very fiddly, the principle that "making such technicalities explicit makes the overall system easier to understand" would make this a virtue – if the subsequent analysis does not degenerate into a mess.

The CC demands that the external interface be specified for all the TSF (or any of their sub-components). This would involve the boundary to the environment, as well as the boundary to other functions (or components), and fits very well with the state-based notion of writing down the system variables. A common state-based system description is the *I/O machine*: the system dynamics is described to the degree of detail required as, for example, a relation between the system's inputs and outputs. This leads naturally to the so-called *dataflow decomposition*, which is a functional decomposition into further I/O sub-components blocks joined by lines representing the state transferred around the system. Ultimately, an I/O component can be replaced (refined to) a state machine where the dynamics are fully specified (the dynamical relation is determined by a function). Again, the CC requirements for functional decomposition make this a very natural approach. In terms of Figure 1 this is "horizontal" development: development within a given layer of the TSF representation hierarchy comprising the ADV class.

A related development, which becomes available in the state-based approach, is "vertical" in the sense of Figure 1: the refinement of the system properties themselves. In this way, the properties induce a hierarchy in which the specification sits at the top. The vertical and horizontal hierarchies can intersect through the sub-component specifications.

From this discussion we may deduce the following additional requirements for the ideal tool to support CC evaluation development.

- The tool must support the description of interacting concurrent systems.
- A requirement of general applicability would tend to favour the state-based approach.
- In that case the tool should support the hook-up of sub-components so that the developer is not mired in trivial but messy analysis.
- The state-based approach is also favoured when the tool supports the dataflow decomposition, producing a "horizontal" system hierarchy which the tool could interact with nicely.
- The tool should support the refinement of system properties, producing a "vertical" hierarchy.
- At least part of the nice interaction with these hierarchies is that the tool should automatically produce documentation which reflects them, with a variety of different views allowed to bring out the best explanations as the setting changes.

The formal verification effort required by the CC then becomes the problem of filling in the arguments which ensure that the steps in the hierarchy are actual refinements. One could in principle carry out such arguments with pen and paper. For the evaluators this is quite disadvantageous, because it forces them to check very low-level logical reasoning. An argument can be very persuasive, but typically a large amount of further detail and reasoning is required to furnish a proof. For example, checking a case analysis of, say, 100 cases with intricate arguments for each would be a huge burden if there was no level at which the details could be trusted to be correct. Thus, we would certainly expect our ideal tool to provide a formal theorem prover.

Proof tools utilise the precision of formal language to implement low-level reasoning via pattern-matching to logical inference rules. The set of rules can be remarkably small, and their consistency established by meta-reasoning. From these rules, quite powerful modelling logics can be constructed. Moreover, the low-level rules can be combined to produce proof tactics tailored to a given modelling paradigm. The evaluators of a proof presented from a proof tool which they understood could trust in the correctness of certain details which they knew the proof tool handled well.

The most useful proof tools along these lines are *interactive*, since they require the user to input each tactic applied in the proof and produce a linear script which summarises the resulting proof. At the lowest level the tool will produce a huge list of low-level tactics for even quite straightforward proofs. The evaluators are then in danger of losing the actual argument, while checking the low-level application. Of course, the advantage to be gained by using the proof tool was that the evaluators could trust the proof tool to handle that level consistently. However, we would argue against blind acceptance of huge collection of tactics with "qed" at the end. The evaluators must be able to see the argument clearly developed. There must be a level of granularity at which the evaluator knows that the statement given is correct because it is true, and the subsequent (probably essentially one line) formal proof is simply part of the tool's book-keeping which ensures the whole result is consistent. At the very least, this layer of granularity should be manifest in the presentation of the verification given by the developers. The ideal tool would support it.

There are some obvious desiderata for the ideal tool support of verification.

- The tool must be based in a formal proof tool.
- The proof tool must then be generic – allowing extensions to treat systems and arguments as they are developed.
- Consistent with our earlier discussion, the entire modelling exercise should be under the control of that proof tool.
- The tool should have well-developed reasoning packages for each modelling paradigm which it supports. These packages would include tactics which work at a variety of levels to ensure the appropriate level of granularity in the presentation of the proofs.
- The tool should provide an interaction to these tactics, and the proof tool in general, which promotes (or almost enforces) the appropriate presentation of the argument.
- The documentation should again reflect this interaction, and display the argument appropriately.
- The tool should reflect when an entire project has been successfully submitted to the proof tool. In particular, there should be normative documentation which builds the entire formal construction in the consistent manner required by the proof tool (as well as that containing the different views discussed earlier).

Part 4

Formal Specification

Formal specification languages can be divided into two main groups: *event-based* and *state-based*. As we will see, state-based languages are a natural setting for describing devices, whilst event-based languages are best used to describe the communication between devices.

Sections 4.1 and 4.2 introduce the respective approaches and give an example of a specification in each to highlight their relative strengths. Section 4.3 discusses current research on integrating the two approaches. Finally, Section 4.4 provides conclusions and further discussion.

4.1 Event-Based Specification

Many systems can be viewed as being composed of several sub-systems or processes that interact concurrently with each other and their environment. Paradigms such as Petri Nets [36] and Process Algebras such as CSP [21, 38], CCS [32], and the pi calculus [33], are notations that take this view towards system specification. Individual processes are modelled as sequences of system *events*, and then multiple processes are specified to interact concurrently in order to achieve the desired behaviour of the entire system.

Security protocols are obvious examples of systems that rely on successful interaction between several communicating processes. An individual instance of a key distribution protocol usually involves two to three processes all working together to establish a secure means of communication between them. The most successful process algebra currently used for modelling security protocols is the CSP approach presented by Ryan et al. [39].

The following example demonstrates the use of CSP and its concise way of capturing security properties using as an example, a simple protocol that transmits a binary acknowledgement that is meant to be kept secret from a potential eavesdropper.

There are three agents: Alice and Bob who are trying to communicate securely, and Eve who is eavesdropping on the communication medium. The sets of events available to each agent process are defined in their *alphabets*. Agent Alice A has two operations s_0 and s_1 representing sending of the 0 and 1 bits respectively, Bob has two operations for receiving the bits, and Eve has two operations for observing the bits.

The alphabets of Alice (A), Bob (B) and Eve (E) are thus:

$$\begin{aligned}&\alpha A = \{s_0, s_1\}\\&\alpha B = \{r_0, r_1\}\\&\alpha E = \{e_0, e_1\}\end{aligned}$$

Alice can either send a 0 and resume the role of process A or ('$\Box$') send a 1 and resume the role of process A. Similarly, Bob can either receive a 0 or a 1 and resume the role of process B, and Eve can either observe a 0 or a 1 and then resume the role of process E. These behaviours are specified below.

$$\begin{aligned}&A = s_0 \rightarrow A \;\Box\; s_1 \rightarrow A\\&B = r_0 \rightarrow B \;\Box\; r_1 \rightarrow B\\&E = e_0 \rightarrow E \;\Box\; e_1 \rightarrow E\end{aligned}$$

We want to ensure that Bob receives transmissions correctly from Alice, and that if Eve is listening, she cannot make sense of the acknowledgement sent. This property is captured in the specification of the channel $CHAN$ below.

$$\begin{aligned}CHAN = s_0 \rightarrow & (r_0 \rightarrow CHAN\\ &\;\Box\; e_0 \rightarrow r_0 \rightarrow CHAN\\ &\;\Box\; e_1 \rightarrow r_0 \rightarrow CHAN)\\ \Box\; s_1 \rightarrow & (r_1 \rightarrow CHAN\\ &\;\Box\; e_0 \rightarrow r_1 \rightarrow CHAN\\ &\;\Box\; e_1 \rightarrow r_1 \rightarrow CHAN)\end{aligned}$$

This means that although Bob always receives Alice's transmissions correctly, Eve always has the option of observing either a 0 or a 1 regardless of what value Alice actually sends.

The entire system is then defined as the concurrent interaction between the three agent processes and the channel process. The parallel operator '$\|$' ensures that processes agree on events that occur.

$$(A \parallel B \parallel E) \parallel CHAN$$

None of Alice's or Bob's events are included in Eve's alphabet and hence do not make up any part of Eve's specification. This means that she cannot agree on any of these events, so she has no way of deriving the transmitted value by observing the channel. Thus, we have abstractly specified the concept of encryption for this particular protocol.

Agent processes and their alphabets must be stated clearly to create the context that makes $CHAN$ secure. If we don't specify that Eve's alphabet consists of only e_0 and e_1 events, then an eavesdropper that can see s_0 and s_1 events,

$$\begin{aligned}E = &\; s_0 \rightarrow e_0 \rightarrow E\\ &\Box s_1 \rightarrow e_1 \rightarrow E\;,\end{aligned}$$

observes the transmission perfectly whilst satisfying the behaviour of $CHAN$.

4.2 State-Based Specification

Alternatively, a system can be viewed as a single 'machine' with an internal state and a set of operations that can be performed on the state in order to change it. Formal notations that take this view towards system specification are usually based on set-theory and logic which means that critical requirements are specified using mathematical predicates that place restrictions on variables and relationships within the state. Examples of such notations are Z [43], Object-Z [18], B [6], and VDM [10], and tools such as DOVE [11]. Z is the most well-known state-based method. The B method [6] is becoming popular because it was created with machine-readability in mind and hence is accompanied by strong tool support for verification.

The author has demonstrated a way in which the Z specification language can be used for specification of security protocols [26, 25]. This method has been used recently to verify a new attack found on what was thought to be a secure version of the Needham-Schroeder security protocol.

Instead of capturing the *concept* of encryption as is demonstrated in the CSP example above, specific properties of the encryption function are made explicit. For example, assuming the existence of a set of plaintext messages PT and ciphertext messages CT, an encryption function and corresponding decryption function could be specified as follows:

$$\begin{array}{|l}
\mathit{encrypt} : PT \rightarrowtail CT \\
\mathit{decrypt} : CT \rightarrowtail PT \\
\hline
\mathit{decrypt} = \mathit{encrypt}^{-1}
\end{array}$$

The functions are injective (as represented by the symbol '$\rightarrowtail$'), which means that each plaintext message has a unique ciphertext representation and vice versa; and because the decrypt function is the inverse of the encrypt function ($\mathit{decrypt} = \mathit{encrypt}^{-1}$), decryption of a ciphertext message will produce the original corresponding plaintext message.

These functions can then be used in a specification of a protocol that transmits encrypted messages. For example, given the following state schema *InTransit* that keeps the value of the ciphertext message *msg* currently in transit,

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{InTransit}} \\
\hline
\mathit{msg} : CT \\
\hline
\end{array}$$

the following *send* operation denotes the transmission of a given message *msg*? in its encrypted form. We must assume that the encryption function is not known to any unauthorised eavesdroppers for the message to remain secret.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{send}} \\
\hline
\Delta \mathit{InTransit} \\
\mathit{msg}? : \mathit{MSG} \\
\hline
\mathit{msg}' = \mathit{encrypt}(\mathit{msg}?) \\
\hline
\end{array}$$

The Z symbol 'Δ' declares the pre-state and post-state variables for the state schema *InTransit*, indicating that the state may change by the operation. Pre-state variables are undecorated and hold the value of the variables before execution of the operation, whereas post-state variables are decorated with a prime and denote the value of the variables after execution of the operation. Therefore, the variable msg' which is specified in the predicate part of the schema contains the encrypted message that is being sent.

A *receive* operation can then be specified to take the message in transit and decrypt it to output the original message $msg!$. (The Z symbol 'Ξ' ensures that state variables remain unchanged.)

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{receive}} \\
\hline
\Xi \mathit{InTransit} \\
\mathit{msg}! : \mathit{MSG} \\
\hline
\mathit{msg}! = \mathit{decrypt}(\mathit{msg}) \\
\hline
\end{array}$$

There is more than one way to specify functional behaviour that achieves *non-functional* (see Section 5.5) security requirements. For example, to state explicitly that the system uses a key-encryption function, we specify the function as follows, producing a unique encryption function for each key (from the assumed set of keys KEY) used:

$$\begin{array}{|l}
\mathit{encrypt} : \mathit{KEY} \rightarrowtail (\mathit{PT} \rightarrowtail \mathit{CT}) \\
\mathit{decrypt} : \mathit{KEY} \rightarrowtail (\mathit{CT} \rightarrowtail \mathit{PT}) \\
\hline
\forall\, k : \mathit{KEY} \bullet \mathit{decrypt}(k) = \mathit{encrypt}(k)^{-1}
\end{array}$$

These functions will also keep transmitted information from being observed by unauthorised parties. It is merely a more explicit variation that requires a different assumption; that is, that eavesdroppers do not know the key used for encryption.

4.3 Integrated Methods

Notations combining both state-based specification languages and process algebras are emerging in order to bring together the best of the two worlds. In particular, there has been substantial research on combining Z and its object-oriented equivalent, Object-Z, with process algebras such as CCS and CSP [19]. For example, Smith has combined Object-Z with CSP, and Mahony and Dong have combined Object-Z with Timed CSP (TCOZ).

There are two significant aspects that differ amongst the resulting combinations. These are

- whether the formalisms have been combined syntactically or semantically, and
- how the process algebra events correspond to the state-based operations.

The language currently looked upon most favourably for its potential use by developers of safety- and security-critical systems in the UK (for example, QinetiQ, DSTL, and Praxis Critical Systems) is *Circus* [48]. It combines Z and CSP at the syntactical level and employs the multi event approach. These design decisions make the language one of the most flexible integration, and also make it a suitable language for refinement, an aspect of formal development in which other combined methods have not been so successful. Mahony and Dong's TCOZ is combined in a similar way.

4.4 Discussion

Sections 4.1 and 4.2 demonstrate, by way of example, the fundamental differences between the approaches taken by the CSP and Z languages. Process algebras are useful for expressing interactions between sub-systems of a larger system and for expressing critical requirements based on how they should interact. At this abstract level the properties can be concise and clever, although sometimes indirect with little indication as to how the requirements could be achieved. On the other hand, state-based methods are not convenient for specifying what sequences of operations are allowed by a system; however, they are extremely useful for expressing functional requirements of specific actions that collectively ensure critical requirements are achieved, without having to give implementation details. Either of the two approaches to specification can be suitable for producing quality specifications, but evidently accentuate different aspects of a system. An integration of the two methods may prove to be most effective; however, this is still a young area of research that must be explored.

Part 5

Formal Verification

Verification may be performed on specifications to prove that certain desirable properties hold. The particular type of verification required by the Common Criteria is a series of correspondence proofs between several models of the one system. Given two such models, a correspondence proof will confirm that certain properties and behaviours specified in one of the models are correctly carried through to the other. This type of verification is known in the formal methods world as a refinement.

The two main approaches to verification are *model checking* and *theorem proving.* Sections 5.1 to 5.4 summarise the advantages and disadvantages pertaining to each and identify some supporting tools, Section 5.5 notes known difficulties associated with refining security functions, and Section 5.6 discusses how model checking and theorem proving would compare when incorporated into a methodology for the Common Criteria.

5.1 Model Checking

A model is an abstraction of a system that simulates the system's behaviour whilst aiming to achieve certain requirements (often temporal) identified by a specification. Given such a model, model checking [12] involves an exhaustive search for every possible trace of operations in order to examine whether the specified requirements always hold. Once the search has begun, it is performed automatically with no need for any user interaction. If the properties hold, the analysis will simply terminate, otherwise a counterexample is presented, providing valuable feedback that helps the developers determine the cause of the problem. This method has been successfully applied to complex industrial designs such as sequential circuits, communication protocols and digital controllers [13].

An exhaustive search is only possible if the state space is finite, otherwise the search would theoretically go on forever — in practice, model checkers run out of space and crash. Furthermore, even with large finite models the checker may not have enough memory to complete the analysis or could run for inconvenient periods of time. These issues are caused by what is known as the *state explosion* problem. To counter this problem, developers must be clever in devising a model that is kind to the model checker used, for example, sometimes an infinite domain must be abstracted into a finite one. This process can be time consuming and can be harmful if information critical for an effective analysis is lost in the abstraction.

One of the most notable advances in the field was the introduction of *symbolic model checking* by McMillan [12]. He realised that by using a symbolic representation for the state transition graphs, much larger systems could be verified. This and other technical advances in the available tools has made model checking more attractive for use in industry. Hence, symbolic model checking has been used by a number of major companies including Intel, Motorola, ATT, Fujitsu, and Siemens for verifying designs [13].

5.2 Model Checking Tools

One of the most popular symbolic model checkers is the Symbolic Model Verifier (SMV) [12] produced by McMillan as a result of his research findings. The SMV system has been widely distributed and has been used for formal verification of a large number of academic and industrial examples. For example, the SMV model checker was applied to a model of the cache coherence protocol described in the IEEE Futurebus+ standard (IEEE Standard 896.1-1991) in order to prove that the system satisfied a formal specification of cache coherence. A number of previously undetected errors were revealed. According to Clarke, this appeared to be the first time that an automatic verification tool had been used to find errors in an IEEE standard [12, pp. 7–8].

The School of Information Technology and Electrical Engineering at The University of Queensland has recently investigated the development of a suite of tools for automatic generation and verification of signalling logic designs for Queensland Rail. The tool used for verification is a new and extended version of SMV called NuSMV. It is used to ensure that the signalling logic satisfies signalling safety principles, for example, that no collisions or derailments can occur.

Other popular model checking tools include Spin, Step, murphi, Java Pathfinder, and μcke [41], and the Failure/Divergences Refinement tool (FDR) [38] which was made specifically to be used in conjunction with CSP.

FDR is the tool used by Ryan and Schneider [39] to analyse their CSP specifications of security protocols. FDR obtains its results by comparing two CSP models, an *implementation* and a *specification*. For checking security protocols, the implementation describes the system in terms of agent and intruder actions, and the specification describes the desired security properties. FDR checks that all possible traces of the implementation are allowed by the specification, that is, that the implementation is a refinement (see Section 5.5) of the specification. If the check fails, a counterexample is presented, revealing those intruder actions that led to the success of the attack. Therefore, given a weak design, failure of the check indicates a need to strengthen the protocol. Alternatively, given a strong design, various mutants (weakenings) can be checked in order to reveal overkill, or to reinforce and make clear the need for certain design decisions [39, pp. 36–37].

5.3 Theorem Proving

In contrast, the theorem proving [41] approach is popular due to its ability to reason with large and even infinite state spaces, as it does not suffer from the state explosion problem. Instead of analysing every possible trace of operations within a system model, a theorem prover conducts verification through inductive reasoning of the system's specification. Therefore, the notation used for the specification is usually a logic with a concrete syntax and semantics with which to reason.

Interactive theorem provers require large amounts of expert user guidance in order to perform the verification proofs; even the smallest of proof steps often need to be applied by the user, and the proofs may take up to weeks or months to complete. To help automate some of these smaller steps, *tactics* are devised that deduce what the prover should do in certain situations. Libraries of such tactics can be produced by users for specific applications. *Automated theorem provers* exist but their languages are restrictive, and they are often only successful for verifying small systems.

5.4 Theorem Proving Tools

One of the most popular theorem provers and the most prominently used in the area of security is the Isabelle theorem prover [34]. It has been used by Paulson [35] to prove desirable properties of security protocols. Using his method, he was able to discover a known attack on the Needham-Schroeder protocol, and in addition he discovered a new attack on a variant of the Otway-Rees protocol. With Bella, he has investigated verification of the Kerberos protocol and verified the SET registration protocols [9].

Another popular theorem prover is the Prototype Verification System (PVS) which has been applied successfully to large and difficult applications in both academic and industrial settings [16]. For example, it has been used to verify the microcode for selected instructions of a complex, pipelined, commercial microprocessor having 500,000 transistors where seeded and unseeded errors were found, and it has also been used as an aid to the requirements analysis of critical, system-level fault protection software on the Cassini spacecraft [28]. Krishnan [37] has used PVS to verify a subset of the SET protocols and NetBill to ensure that they achieve certain secrecy properties.

The University of Queensland has developed a theorem prover called Ergo. It provides us with the ability to verify specifications for correctness, and also to refine specifications into programs. Ergo has a Z theory for reasoning with Z specifications and a B theory is currently being devised. Johnston [23] has used the Ergo theorem prover to develop a cryptographic protocol by stepwise data refinement from a Z specification.

Tools such as Isabelle, PVS and Ergo, are interactive theorem provers in their own right. That is, the theorem provers' notations are general and not restricted to a particular method of specification. Therefore, it is often the case that a conversion (manual or automatic) is needed to produce the input language for the desired prover from the specification. Purpose built theorem provers also exist to support particular specification languages, such as Z/EVES [40] and ProofPower [24] for Z, and the theorem prover provided within the B toolkit [6]. Woodcock and Cavalcanti [48] state that *Circus* models can be analysed with almost no changes using Z/EVES.

DSTO's DOVE [11] tool provides a graphical environment for specification and subsequent verification of systems. Specifications are primarily produced by constructing state machine diagrams. However, whilst building the diagrams, the user can supply information regarding states and transitions, aiding the parallel construction of a lower level formal model that can be verified using the integrated Isabelle theorem prover. The tool also provides automatic generation of documentation to accompany the formal specifications and proofs. This aspect is particularly useful for providing evidence to independent third parties such as those needed for security evaluations. DOVE was successfully used for developing the Starlight security products that have been evaluated to ITSEC E6 (see Section 6.3). It has also been used by McCarthy [31] to analyse security in the context of classified rooms.

5.5 Refining Security Properties

There are two types of system requirements: functional and non-functional requirements. Functional requirements are constraints on the behaviour of operations, whereas non-functional requirements are constraints on the results of one or a series of operations, or on the system as a whole [14]. Properties referred to as security properties are generally non-functional. The problem with this is that functional refinement does not necessarily preserve non-functional security properties [29, 20]. Lowe [27] demonstrates this using a simple CSP example given in Section 5.5.1.

Collinson [14] and CESG's Manual "F" [7, 15] present an approach using Z in which the functional specification of a system is produced, and then the security model is expressed in terms of this specification. Unfortunately, it has since been revealed that the proofs required are too complex and that Z is inappropriate for modelling non-functional security properties [47].

Logica used Z to develop an electronic cash system for smart cards called Mondex, and the supporting operating system called Multos [44]. These systems were evaluated against the ITSEC and certified to level E6. All but one of the security requirements were functional requirements. This meant that Z was sufficient for almost the entire development because functional requirements are preserved by refinement. The one non-functional security requirement concerning information flow was captured using Z, with additional CSP to constrain allowed traces of events. In order to refine this property, an *unwinding theorem* was used to derive equivalent properties of individual state transitions [45].

5.5.1 Lowe's Refinement Example

Lowe [27] demonstrates that non-functional security properties are not necessarily preserved by refinement using a simple CSP example.

There are two agents: Hugh who works in a high security domain, and Lois who works in a low security domain. Hugh's alphabet (the set of events visible to Hugh)

$H = \{h\}$ consists of one high event h, and Lois' alphabet $L = \{l\}$ consists of one low event l. Some systems insist that no information about events performed in a high security domain should be observable by agents residing in a low security domain. This is a *non-interference* property.

In such a system, a process P_1 that performs event h followed by event l and then stops,

$$P_1 = h \rightarrow l \rightarrow STOP \ ,$$

is insecure since Lois' event can only occur after Hugh's, hence she can always identify when Hugh has performed an event. However, a process P_2 that performs event h followed by event l *or* ($\Box$) only event l and then stops,

$$P_2 = (h \rightarrow l \rightarrow STOP) \ \Box \ (l \rightarrow STOP) \ ,$$

is secure since l can always occur regardless of h revealing no information to Lois about Hugh's actions.

Consider now a process Q_1

$$\begin{aligned} Q_1 = \ & h \rightarrow (l \rightarrow STOP \ \sqcap \ STOP) \\ & \Box \\ & (l \rightarrow STOP \ \sqcap \ STOP) \end{aligned}$$

that will non-deterministically '$\sqcap$', either allow Lois to perform event l or will have Lois' event refused, whether or not Hugh performs an h.

Like P_2, Q_1 should be secure. However, if the first non-deterministic choice is implemented to always select its first argument, and the second non-deterministic choice is implemented to always select its second argument, then Q_1 will act like

$$\begin{aligned} & h \rightarrow l \rightarrow STOP \ \Box \ STOP \\ = \ & h \rightarrow l \rightarrow STOP \\ = \ & P_1 \ . \end{aligned}$$

So although Q_1 is considered secure, one of its refinements is not.

5.6 Discussion

Because model checking can be performed automatically, it is often considered to be preferable to theorem proving whenever it can be applied [12, p. 3]. However, model checkers are unable to produce a formal proof that can be checked externally. If a model does not satisfy the specification, a counterexample is presented. But if it does, the tool simply terminates and the user must trust the implementation of the tool [41, p. 6].

Some form of formal verification is required to adhere to the requirements set out in the Common Criteria. If theorem proving is used, the evaluators could be provided with a proof script that they themselves may follow for satisfaction in the verification. On the other hand, if model checking is used, either the evaluators would have to be confident in the ability of the tool, or they could perform some checks of their own on mutants (weakenings) of the models. These tests should present counterexamples that may convince the evaluators that the original models were correctly verified; however, it would not be feasible for evaluators to check every possible mutation.

Part 6

Examples

A wide range of security-enforcing hardware and software products have been evaluated at varying levels spanning several criteria. In the following sections we list summarise what formal methods are used by various organisations for developing security products to achieve high assurance levels. There are probably more examples that would be relevant; however, due to confidentiality restrictions they are not easily accessible.

6.1 Naval Research Laboratories

The Naval Research Laboratory (NRL) uses a formal method called SCR for developing security products to high assurance levels. SCR is a tabular formal method used for specifying and analysing operational requirements of safety-critical control systems,

Kirby, Jr. et al. [22] applied NRL's SCR requirements method to a US Navy COMSEC device that provides cryptographic processing for a US Navy radio receiver. A subset of the informal requirements specification was translated into a formal SCR specification and then several analyses were performed using the SCR* toolset revealing some weaknesses.

The SCR* toolset provides several analysis techniques for verification of an SCR specification, including both model checking and theorem proving. Each type of analysis has advantages and disadvantages. Therefore, applying several types of analyses helps to identify different types of flaws.

6.2 Logica

MULTOS [30] is an operating system specifically designed to exist on a smartcard and Mondex is an electronic cash application that runs on MULTOS allowing smart cards to act exactly like cash, offering immediate transfer of value without signature, PIN, or transaction authorisation. These applications were developed by Logica using the Z specification language and were evaluated to ITSEC level E6 [44] (see Section 5.5).

6.3 Defence Science & Technology Organisation

The Starlight Interactive Link [17] security devices — consisting of a data diode, a multiple computer switch, and a key board switch — were developed by the Defence Science & Technology Organisation to aid information security in a multi-level security environment.

The Z notation was initially used for the formal specifications, however, there was little state to model and the security properties were based on information flow (a non-interference policy model) rather than state. Therefore, there was little to be gained by a state machine view, so the main modelling for both the policy model and the architecture was carried out using the HOL logic of the Isabelle theorem prover. The DOVE tool was used as an interface to document the proofs.

6.4 Compucat

Compucat has developed a Serial Data Regulator (SDR) or data diode, and a secure X.25 Packet Assembler-Disassembler (CSX-100) and both have been evaluated to ITSEC level E6. The formal methodology applied used set theory in conjunction with a modelling approach based upon the work of Bell and La Padula [8].

Part 7

Conclusion

The Common Criteria demands formal methods to be used for the development of security products in order for them to be evaluated to the highest assurance levels. Formal specification is required to produce precise descriptions of the system and verification is required to prove that all specifications correspond to each other. Currently there is no standard methodology in place to achieve this.

The two most popular approaches to specification are event-based and state-based methods. Event-based languages are convenient for describing systems consisting of concurrently running processes and for stating non-functional properties regarding how such processes are to interact with each other. Whereas state-based methods are convenient for describing the specific functional properties of individual operations that collectively ensure critical non-functional requirements are achieved. Researchers are currently investigating integrations of both approaches to bring together the best of the two worlds.

Model checking and theorem proving are the two most popular approaches to formal verification. Model checking provides an automatic but invisible verification of finite models that requires significant user expertise in producing models in a way so as to reduce the state explosion problem. Whereas theorem proving provides a manual and visible verification of infinite models requiring significant user expertise in performing the proofs.

Security properties can be difficult to refine as they are often expressed in a non-functional way. It has been shown that a security policy consisting of functional security properties is more successfully refined than a security policy consisting of non-functional security properties.

Formal methods currently used for formal specification and verification of security products to achieve high levels of assurance are SCR used by NRL, Z used by Logica, and DOVE used by DSTO.

Acknowledgements

This report was funded by the Australian Defence Science and Technology Organisation under a Research Agreement. I would like to thank Jim McCarthy, Luke Wildman, Colin Fidge, Brendan Mahony, Tony Cant, Peter Robinson and Andrew Rae for useful discussions and for comments on earlier versions of this report, and also Rod Chapman, Gary Carlisle, Anthony Hall, Jim Kirby, Susan Stepney, Chris Walsh, Kylie Williams, and various people from DSTL, QinetiQ and NSA for their time helping us with queries on this topic.

References

1. Common Criteria for Information Technology Security Evaluation. August, 1999. Version 2.1. CCIMB-99-031. http://csrc.nist.gov/cc/.

2. Trusted Computer System Evaluation Criteria. December 26, 1985. DoD 5200.28-STD.

3. Information Technology Security Evaluation Criteria (Version 1.2). Printed and Published by the Department of Trade and Industry, London, June 1991. http://www.itsec.gov.uk/.

4. The Canadian Trusted Computer Product Evaluation Criteria. Canadian System Security Centre, Communications Security Establishment, Government of Canada. Version 3.0e, January 1993.

5. Australasian Information Security Evaluation Program (AISEP), Publication No 1, 2001.

6. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, UK, 1996.

7. R. Barden, S. Stepney, and D. Cooper. *Z in Practice*. BCS Practitioner Series. Prentice Hall, UK, 1994.

8. D.E. Bell and L.J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report ESD-TR-75-306, The Mitre Corporation, 1976.

9. G. Bella, F. Massacci, and L. C. Paulson. Verifying the SET registration protocols. *IEEE Journal On Selected Areas In Communications*, 21(5):77–87, January 2003.

10. D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

11. T. Cant, B. Mahoney, and J. McCarthy. Design oriented verification and evaluation: the dove project. Technical Report DSTO-TR-1349, Defence Science and Technology Organisation, December 2002.

12. E.M. Clarke, Jr., O. Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, London, 1999.

13. E.M. Clarke and H. Schlingloff. Model checking. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 24, pages 1635–1790. Elsevier Science, 2001.

14. R. Collinson. A critical look at functional specifications. In M. Naftalin, B. T. Denvir, and M. Bertran, editors, *Second International Symposium of Formal Methods Europe (FME 1994)*, volume 873 of *Lecture Notes in Computer Science*, pages 381–400, Germany, 1994. Springer.

15. Computer and Electronic Security Group. CESG computer security manual "F": A formal development method for high assurance systems. Issue 1.1, July, 1995.

16. Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, , and Mandayam Srivas. A tutorial introduction to PVS. April 1995. Accessed 2004. http://www.csl.sri.com/papers/wift-tutorial/.

17. Defence Science & Technology Organisation. Starlight world class multi-level information security. Accessed 2003. www.dsto.defence.gov.au/isl/starlight.pdf.

18. R. Duke and G. Rose. *Formal Object-Oriented Specification Using Object-Z*. Cornerstones of Computing. Macmillan Press Limited, UK, 2000.

19. C. Fischer. How to combine Z with a process algebra. In J.P. Bowen, A. Fett, and M.G. Hinchey, editors, *11th International Conference of Z Users (ZUM '98)*, volume 1493 of *Lecture Notes in Computer Science*, pages 5–25, Germany, 1998. Springer-Verlag.

20. J. Graham-Cumming and J. W. Sanders. On the refinement of non-interference. In *Computer Security Foundations Workshop IV, 1991*, pages 35–42. IEEE Computer Society, 1991.

21. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

22. J. Kirby, Jr., M. Archer, and C. Heitmeyer. Applying formal methods to an information security device: an experience report. In *Proceedings of the 4th IEEE International Symposium on High-Assurance Systems Engineering*, pages 81–88, November 1999.

23. W. Johnston. Development of a cryptographic protocol by stepwise refinement using the cogito methodology: Part 1. Technical Report 99-38, Software Verification Research Centre, January 2000.

24. D. J. King and R. D. Arthan. Development of practical verification tools. *ICL Systems Journal*, 11(1), May 1996.

25. B. W. Long. Formal verification of type flaw attacks in security protocols. In *Proceedings of the 10th Asia-Pacific Software Engineering Conference (APSEC) 2003*, pages 415–424. IEEE Computer Society, 2003.

26. B. W. Long, C. J. Fidge, and A. Cerone. A Z based approach to verifying security protocols. In J. S. Dong and J. Woodcock, editors, *5th International Conference on Formal Engineering Methods, ICFEM 2003*, volume 2885 of *Lecture Notes in Computer Science*, pages 375–395. Springer-Verlag, 2003.

27. G. Lowe. Defining information flow. Technical Report 1999/3, Department of Mathematics and Computer Science, University of Leicester, 1999.

28. R. Lutz and Y. Ampo. Experience report: Using formal methods for requirements analysis of critical spacecraft software. In *Proceedings of the 19th Annual Software Engineering Workshop*, Greenbelt, MD, 1994.

29. H. Mantel. Preserving information flow properties under refinement. In *IEEE Symposium on Security and Privacy, 2001*, pages 78–91. IEEE Computer Society, 2001.

30. MAOSCO Limited. Welcome to MULTOS. Accessed 2003. www.multos.com.

31. J. McCarthy and J. Thredgold. Modelling smart security for classified rooms with DOVE. In C. Lakos, R. Esser, L. M. Kristensen, and J. Billington, editors, *Proceedings of the Workshops on Software Engineering and Formal Methods and Formal Methods Applied to Defence Systems*, volume 12 of *Conferences in Research and Practice in Information Technology*, pages 135–144. Australian Computer Society, 2002.

32. R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science. Springer-Verlag, 1980.

33. R. Milner. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, May 1999.

34. T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, Germany, 2002.

35. L. C. Paulson. Proving properties of security protocols by induction. In *Proceedings of 10th IEEE Computer Security Foundations Workshop (CSFW'97)*, pages 70–83. IEEE Computer Society Press, 1997.

36. J. L. Peterson. *Petri Net Theory and The Modeling of Systems*. Prentice Hall, 1981.

37. A. Renaud and P. Krishnan. An environment for specifying and verifying security properties. In *Proceedings of the Australian Software Engineering Conference*, pages 203–212. IEEE Computer Society Press, 2001.

38. A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall Series in Computer Science. Prentice Hall, 1998.

39. P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *The Modelling and Analysis of Security Protocols: The CSP Approach*. Addison-Wesley, 2000.

40. M. Saaltink. The Z/EVES system. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *10th International Conference of Z Users (ZUM '97)*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–85. Springer, 1997.

41. J.M. Schumann. *Automated Theorem Proving in Software Engineering*. Springer, Germany, 2001.

42. Richard E. Smith. Trends in government endorsed security product evaluations. In *Proceedings of the 23rd National Information Systems Security Conference*, Baltimore, United States, October 2000.

43. J. M. Spivey. *The Z Notation : A Reference Manual*. Prentice Hall International Series In Computer Science. Prentice Hall, London, 1992.

44. S. Stepney. New horizons in formal methods. *The Computer Bulletin*, pages 24–26, January 2001.

45. S. Stepney. Formal development of security-critical smart card applications, May 2003. Accessed 2003. http://www.csr.ncl.ac.uk/resources/164/SusanStepney.pdf.

46. R. C. Summers. *Secure Computing: Threats and Safeguards*. McGraw-Hill, USA, 1997.

47. J. Woodcock. Software engineering research directions. *ACM Computing Surveys*, 28(4es), December 1996.

48. J. C. P. Woodcock and A. L. C. Cavalcanti. A concurrent language for refinement. In *5th Irish Workshop on Formal Methods*, 2001.

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION DOCUMENT CONTROL DATA			1. CAVEAT/PRIVACY MARKING	
2. TITLE The Role of Formal Methods in High-Grade InfoSec Evaluations			3. SECURITY CLASSIFICATION Document (U) Title (U) Abstract (U)	
4. AUTHOR Benjamin W. Long School of Information Technology and Electrical Engineering The University of Queensland Brisbane, Australia 4072			5. CORPORATE AUTHOR Defence Science and Technology Organisation PO Box 1500 Edinburgh, South Australia 5111, Australia	
6a. DSTO NUMBER DSTO–GD–0456	6b. AR NUMBER 013-595		6c. TYPE OF REPORT General Document	7. DOCUMENT DATE March, 2007
8. FILE NUMBER	9. TASK NUMBER JTW 02/106	10. SPONSOR AS INFOSEC DSD	11. No OF PAGES 37	12. No OF REFS 48
13. URL OF ELECTRONIC VERSION http://www.dsto.defence.gov.au/corporate/reports/DSTO–GD–0456.pdf			14. RELEASE AUTHORITY Chief, Information Networks Division	
15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT *Approved For Public Release* OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SOUTH AUSTRALIA 5111				
16. DELIBERATE ANNOUNCEMENT No Limitations				
17. CITATION IN OTHER DOCUMENTS No Limitations				
18. DSTO RESEARCH LIBRARY THESAURUS High assurance systems Computer security Evaluation Information systems security Technology assessment				
19. ABSTRACT With the increasing use of computer systems in governmental, commercial and industrial equipment, we must be sure that these systems remain secure. The Common Criteria is an internationally recognised criteria for evaluating IT products with security functionality. To achieve a high level of assurance from the Common Criteria, formal methods should be applied in the development process. This report concentrates on formal methods support for development and evaluation of security-critical systems in the Common Criteria. In particular, the Defence Signal Directorate (DSD) is charged with the oversight of the security evaluations program in Australia. The report attempts to indicate what DSD should know about formal methods for the high-grade evaluations, and where they can find out more as desired.				